United States Patent Application

for

METHOD AND MECHANISM FOR EFFICIENT STORAGE AND QUERY OF XML DOCUMENTS BASED ON PATHS

Inventors:

Ravi Murthy

Eric Sedlar

Nipun Agarwal

Prepared By:

Peter C. Mei
Bingham McCutchen LLP
Three Embarcadero Center, Suite 1800
San Francisco, California 94111
(650) 849-4870

Assignee: Oracle International Corporation
500 Oracle Parkway
Redwood Shores, CA 94065

CROSS-REFERENCE TO RELATED APPLICATIONS

The present application claims priority to U.S. Provisional Application Serial No. 60/500,450, filed on September 5, 2003, which is hereby incorporated by reference in its entirety.

COPYRIGHT NOTICE

BACKGROUND AND SUMMARY

The extensible markup language (XML) is a meta-language developed and standardized by the World Wide Web Consortium (W3C) that permits use and creation of customized markup languages for different types of documents. XML is a variant of and is based on the Standard Generalized Markup Language (SGML), the international standard meta-language for text markup systems that is also the parent meta-language for the Hyper-Text Markup Language (HTML).

Since its adoption as a standard language, XML has become widely used to describe and implement many kinds of document types. Increasingly greater amounts of content are being created and stored as XML documents in modern computing systems, with the XML documents often being stored in database management systems. Therefore, there is a growing demand for database systems that provide capabilities to store, manage and query XML content natively in a database. As such, mechanisms for efficient storage and querying of arbitrary XML data is becoming important in building a scalable and robust content management platform.

The content of XML documents may be structured or unstructured. Structured data will conform to an XML schema. Unstructured data may not be associated with any specifically identifiable schema. For example, unstructured XML documents may be created as a result of ad hoc editing. As another example, an unstructured XML document may be created by combining multiple structured documents together into an unstructured collection. There are many scenarios in which users need to store and query XML documents that do not conform to any pre-defined XML schemas.

One of the severe limitations of conventional databases that work with XML data is the lack of efficient processing for schema-less XML documents, particularly when attempting to perform XPath processing on these schema-less documents. XPath is a language for addressing parts of an XML document that has been defined by the W3C organization, in which the parts of an XML document are modeled as a tree of nodes. Further information about the XPath language can be found at the W3C website at http://www.w3.org/TR/xpath, the contents of which are incorporated herein by reference in its entirety. Queries involving XPath predicates are often used to filter XML documents and extract fragments within documents.

In many cases, documents that do not conform to an XML Schema can only be stored in CLOB columns. However, this mode of storage impacts the performance of

XPath-based searches. Inverted indexes and functional indexes can be used to improve certain types of filter queries. However, the more general form of filter queries which involve range predicates and collection traversals are still not satisfied by such indexes, and hence require inefficient DOM-based evaluation. Moreover, functional indexes can be built only on XPath expressions returning a single value. If the XPath expression returns more than one value, a functional index cannot be created. An inverted list index serves as a primary filter but needs an expensive functional evaluation of the XPath as a post-filter operation. The post-filter step is a significant bottleneck especially for large documents. Finally, neither of the two indexing options are effective in extracting fragments based on user specified XPaths.

Embodiments of the present invention disclose a new approach for storing, accessing, and managing data, such as XML data. Also disclosed are embodiments of new storage formats for string XML data. The approach supports efficient evaluation of XPath queries and also improves the performance of data/fragment extraction, and can be applied to schema-less documents. The invention is applicable to all database systems and other servers which support storing and managing XML content. In addition, the approach can be applied to store, manage, and retrieve other types of unstructured or semi-structured data in a database system.

Further details of aspects, objects, and advantages of the invention are described below in the detailed description, drawings, and claims. Both the foregoing general description and the following detailed description are exemplary and explanatory, and are not intended to be limiting as to the scope of the invention.

BRIEF DESCRIPTION OF THE DRAWINGS

The accompanying drawings are included to provide a further understanding of the invention and, together with the Detailed Description, serve to explain the principles of the invention. The same or similar elements between figures may be referenced using the same reference numbers.

Fig. 1 is a flowchart of a process for managing XML data according to an embodiment of the invention.

Fig. 2 is a flowchart of a process for storing XML data according to an embodiment of the invention.

Figs. 3a-c shows an example XML document.

Fig. 4 shows an example Path_Table according to an embodiment of the invention.

Fig. 5 shows an example Path_Index_Table according to an embodiment of the invention.

Fig. 6 is a flowchart of a process for converting an XPath expression to a SQL query according to an embodiment of the invention.

Fig. 7 is a diagram of a computer system with which the present invention can be implemented.

DETAILED DESCRIPTION

Embodiments of the present invention provide methods, systems, and computer program products for managing, storing, and accessing unstructured and semi-structured data, such as XML documents, in relational and object-relational database systems. For the purpose of explanation, the following description is specifically made with reference to managing, storing, and accessing XML documents in a relational database system. It is noted, however, that the following description is equally applicable to other types of data in other types of storage systems, and is not to be limited in its scope to only XML documents. In addition, the following description explicitly uses query syntax conforming to the structured query language (SQL). It is noted that the following description is also applicable to other types of query languages and syntaxes.

Embodiments of the present invention discloses a new approach for storing, accessing, and managing XML data which supports efficient evaluation of XPath queries and also improves the performance of data/fragment extraction. The approach can be applied to schema-less documents to enable efficient XPath processing. In one embodiment, the approach is similar to name-value pair storage but extended to handle mapping from paths to values – without losing the hierarchical (parent-child) information. Some advantages of this approach include more generic solution to store any XML collection, efficient and/or exact filtering for a large subset of XPath expressions, and more usefulness for extracting fragments based on XPath expressions.

An embodiment provides an efficient mechanism for storing arbitrary XML data (not conforming to any schema) based on paths. This storage mechanism allows for high performance of evaluating a large class of XPath queries including range predicates and collection traversals. These benefits easily outweigh the possible increase in the time to insert the document, e.g., due to the overhead of shredding the document into multiple rows, and to reconstruct the entire document, e.g., due to the extra work in putting together multiple relational rows into a document.

As noted, embodiments of the invention provides an approach to define format(s) for storing, accessing, and managing arbitrary XML data comprising sets of documents not conforming to any schema. Fig. 1 is a flowchart showing a high-level overview of a process for storing, accessing, and managing XML documents in a relational database system. At 102, the process begins by storing the XML document(s) into a defined relational schema.

In one approach, the format for storing the XML data can be configured based upon a data model that may be either commonly defined or desirable for processing purposes. For example, in one embodiment, the storage format can be configured to facilitate XPath processing, and therefore the storage format can be defined based upon the tree-of-nodes approach for modeling XML documents that is specified by the XPath standards. At 104, decisions may be made regarding whether to create one or more indexes upon the stored XML data. If so, then the index could be created corresponding to the fields of the defined schema format for storing the XML data (106). Once the XML data has been stored into the desired storage formats, operations can be performed to access the stored XML data (108). Each of these process actions is described in more detail below.

As mentioned above, the format for storing the XML data can be configured based upon a defined data model, such as a storage format that is configured to facilitate XPath processing. Described here is one embodiment of a storage format for storing XML data that is defined based upon the tree-of-nodes approach for modeling XML documents, e.g., as specified by the XPath standards. In this embodiment, any set of arbitrary XML documents, e.g., an XML collection, can be stored in a single (universal) relational schema consisting of two tables. The first table, referred to herein as the PATH_TABLE, stores the path, value pair and associated hierarchical information for the XML data. The second table, referred to herein as the PATH_INDEX_TABLE, assigns unique path ids to path strings, thereby avoiding repeated storage of large path strings. It is noted that the second table is not required, and that the full path can be stored within the PATH_TABLE. However, this approach may be less efficient since it may cause the same large path strings to be repeated multiple times within the table.

The following shows an example schema for the PATH_TABLE according to an embodiment of the invention:

Column Name	Column Type	Description
DOCID	NUMBER	Unique ID for a document
PID	NUMBER	Unique ID for a path (key to PATH_INDEX_TABLE)
STARTPOS	NUMBER	Starting position (pre-order number)
ENDPOS	NUMBER	Ending position (post-order number)
NODELVL	NUMBER	Depth of the node
NODETYPE	NUMBER	Type of the node – element/attribute/text/…
NODEVAL	VARCHAR2(4000)	Value of the node if attribute/text/…

In this schema, the DOCID refers to the document identifier that is assigned to the XML documents. Each XML document will have a unique DOCID value. PID refers a unique identifier for a path, which functions as a key into the PATH_INDEX_TABLE. Multiple nodes within an XML document may have the same path, and therefore may be associated with the same PID value. In the present embodiment, a "node" can be defined as specified in the standard XPath specifications from W3C. The STARTPOS entry identifies the starting position of a node and the ENDPOS entry identifies the ending position of that node. Based on pre-order and post-order traversal of the tree of nodes, the NODEVALVL entry identifies the hierarchical level of a node within an XML document. The NODETYPE column identifies the type of the node that is associated with the present entry. Examples of such types could include an element type, attribute type, or text type. In one embodiment, these types are implemented to be similar to the node types defined by the DOM standard. If the node associated with the present entry is associated with a value, e.g., because the node is an attribute or text type, then the NODEVAL column will contain the node value.

The following shows an example schema for the PATH_INDEX_TABLE according to an embodiment of the invention:

Column Name	**Column Type**	**Description**
PID	NUMBER	Unique ID for the path
PATH	VARCHAR2(1400)	Path string
NODENAME	VARCHAR2(1000)	Name of terminal node

Each PID entry identifies a unique path. The PATH column stores the path value that is associated with a PID. The NODENAME column identifies the terminal node for a given path. The NODENAME column can also be defined as a virtual column base upon the PATH column. This type of column is useful while reconstructing the document/fragment to create the appropriate tag names.

Fig. 2 shows a flowchart of an embodiment of a process for storing XML documents into a database using these tables. When an XML document is processed for storage, a unique identifier is assigned to that document, i.e., in the DOCID field. If a separate PATH_INDEX_TABLE is not used, then the entire path for a given node is stored in the PATH_TABLE (218).

If both a PATH_TABLE and a PATH_INDEX_TABLE are used, then the entire path for a node is stored in the PATH_INDEX_TABLE and only a path identifier for that node is stored in the node entry in the PATH_TABLE. In this approach, the path associated with the node is identified at 206. A determination is made whether an entry for the identified path already exists in the PATH_INDEX_TABLE (208). If so, then the identifier for the path is identified (210) and associated with the node (216), i.e., by storing the PID value in the PID column for the node. Otherwise, a new PID value is assigned (212) and a new entry is created in the PATH_INDEX_TABLE for the newly identified path (214). The new PID value is thereafter associated with the node in the PID column of the PATH_TABLE (216).

At 220 and 222, hierarchical information and type/value information for the node is stored in the entry for the node in the PATH_TABLE. The hierarchical information for the XML data is tracked by viewing the XML document as a tree and assigning a start and an end position to each node, e.g., by using pre-order and post-order traversal numbers. In addition, the node level (tree depth) and the node type are stored. Node values are stored for leaf text nodes, attribute nodes, and other nodes that are associated with a value.

At 224, a determination is made whether there are further node(s) to process within the XML document. If so, then the process returns back to 204 to process the next node within the XML document. Otherwise, at 226, a determination is made whether there are further XML document(s) to store in the database. If so, then the process returns back to 202 to process the additional XML documents.

To illustrate this process, consider the example XML document 300 shown in Fig. 3a. A number of different nodes are present in this document. For the purposes of explanation, consider if the start, end, and attribute value portions of each element in the document are assigned to a position number. These position number will then be used to define the relative start positions and end positions for the nodes or fragments in the document. Fig. 3b shows the position numbers for each of the document portions in the example XML document 300. Fig. 3c shows how the position numbers can be defined by identifying pre-order and post-order traversal numbers for a tree model of the document. In this example document 300, element "a" is at the highest level of the document hierarchy and begins at position 1 and ends at position 19.

Fig. 5 shows an example Path_Table 500 for the XML document 300 of Figs. 3a-c. Consider the first entry 502 in this table, which corresponds to the "a" element in the document 300. Assume that the DOCID value of "1" has been assigned to the XML document 300. Therefore, the DOCID column of entry 502 contains this value of "1". It is noted that all entries in the Path_Table 500 associated with the same XML document 300 will have the same DOCID value.

The "PID" value provides a key into the Path_Index_Table to find the correct path associated with an entry in the Path_Table 500. For entry 502, the PID value of "1" corresponds to a path of "a.". Fig. 4 shows an example Path_Index_Table 400 that is associated with the XML document of Fig. 3a. Entry 402 in Path_Index_Table 400 includes a PATH column that contains the actual path value associated with the PID. One advantage of having this type of table is that the same pathnames do not have to be repeated over and over again to reference the different nodes in the XML document. Instead, the different nodes can be associated with the appropriate PID in this table to be associated with the correct path within the document.

Referring back to Fig. 5, the "STARTPOS" column for entry 502 identifies the start position for element "a", which is the position of the <a> node. Here, it begins at the first position of the document, hence having a position of "1". The "ENDPOS" value identifies the ending position of the "a" element, which is at the position of the </a> node. Here, it ends at the last position of the document, and when each position in this example document 300 is counted if it is a start node, end node, or attribute node, then the end position for this element is at position 19.

The NODELVL column identifies the hierarchical level of an element. Element "a" is at the highest hierarchical level of the XML document 300, and therefore is associated with a value of "1" in the NODELVL column for entry 502.

The NODETYPE column identifies the type of node that is being stored. Here, entry 502 corresponds to element "a" , and therefore the node type stored in the NODETYPE column for entry 502 would be of type "element". The contents within the NODETYPE column can also be stored as numerical equivalents defined for each type, e.g., ELEMENT=1, ATTR=2, TEXT=3, etc.

The NODEVAL column stores the node value, if any, that is associated with the entry. Here, element "a" is not directly associated with a node value. Therefore, the NODEVAL column for entry 502 does not contain a stored value.

The other entries in Path_Table 500 similarly define the other portions of the XML document 300. The other entries in the Path_Index_Table 400 define the other paths that appear in the XML document 300.

One or more indexes can be created on the Path_Table 500 and Path_Index_Table 400 to speed up the evaluation of XPath queries and document and fragment construction operations. For example, to improve the efficiency of the document retrieval and XPath processing, the following are examples of indexes, e.g., Btree indexes, that can be created on the PATH_TABLE 500 in one embodiment of the invention:

- pid
- docid, startpos
- docid, nodelvl, startpos
- substr(nodeval, 1, 1400)

The following are examples of indexes that can be created on the PATH_INDEX_TABLE 400 in one embodiment of the invention:

- unique index on pid [primary key]
- unique index on (path) reverse

Once the XML document has been stored into this type of schema, all or part of the document can be accessed by querying against the known columns of the stored version of the document. In this manner, any of the well-known query methods that have been extensively provided to access relational database tables can be used to efficiently and effectively access XML data stored with this approach. For example, the structured query language (SQL) is a widely adopted mechanism for accessing data stored in a relational database system. The presently described embodiment of the invention provides an approach for allowing SQL to be used to query, access, and reconstruct the stored XML data, even if the XML data was originally unstructured or semi-structured.

A document can be reconstructed very efficiently in a streaming fashion by evaluating the following example SQL query. The query returns all the nodes of the XML document (identified by a docid value) in the document order. Based on the start, end positions and the node level, the appropriate tagging can be added to the output XML stream.

```
select i.nodename, p.startpos, p.endpos, p.nodetype, p.nodeval
from path_table p, path_index_table i
where p.docid = :1 and p.pid = i.pid
order by p.startpos
```

A fragment can be identified by a rowid of the row in the path_table corresponding to the element. Given a rowid of the path_table, the corresponding fragment can be constructed by evaluating the following query. The query returns the nodes within the fragment in document order. Based on the start, end positions and the node level, the output fragment can be constructed.

```
select i.nodename, p.startpos, p.endpos, p.nodetype, p.nodeval
from path_table p, path_index_table i,
  (select docid, startpos, endpos from path_table
   where rowid = :1) p2
where p.docid = p2.docid and p.startpos >= p2.startpos
and p.endpos <= p2.endpos and p.pid = i.pid
order by p.startpos
```

As noted above, one of the current limitations of prior database systems is the lack of efficient XPath processing for schema-less XML documents. The primary syntactic construct in the XPath language is the XPath expression. An XPath expression is evaluated to yield an object, which corresponds to the result of a search upon one or more XML documents.

Embodiments of the present invention provide an approach for facilitating and enabling XPath processing. This section describes how XPath expressions are translated into queries on the underlying path and index tables corresponding to XML documents stored as described with respect to Figs. 1-5. Using this approach, any XPath expression can be converted into a SQL query to access the stored XML data. Fig. 6 shows a flowchart of an embodiment of a process for rewriting an XPath expression into a SQL query.

At 602, the process breaks the input XPath expression into multiple components, e.g., using the following rules:

1. Each continuous segment of simple XPath, e.g., a set of node names separated by "/" such as /a/b/c/d, corresponds to a single XPath component. The term "a//b" means any b that is a child of a, but at level of the hierarchy.
2. Each occurrence of a predicate within the XPath causes creation of new components. For example, /a/b[@id="2"]/c/d consists of the following components - /a/b and @id and /a/b/c/d.

At 604, the process creates a SQL query corresponding to each of the XPath components. The SQL query comprises a join of the path_index_table and the path_table and further includes, for example, the following :

1. Condition for the path being chosen;
2. Condition for the node type (if needed); and/or
3. Condition for the node value (if present).

At 606, the process joins the SQL query corresponding a component to its previous component using, for example, the following join conditions:

1. Join on the docid (i.e., for the same document); and/or
2. Join on the hierarchy relationship – startpos, endpos (e.g., a parent-child relationship).

The next section of this document describes several examples of the embodiment of the translation techniques to convert an XPath expression into a SQL query.

The following example XPath expression searches for the content(s) of one or more XML fragments corresponding to the location path "/a/b/c/d".

```
XPath: /a/b/c/d
```

In the XPath language, a relative location path consists of a sequence of one or more location steps separated by the "/" symbol. The steps in a relative location path are composed together from left to right.

Using the process described above, this XPath expression can be translated to the following SQL statement that queries against the Path_Table and Path_Index_Table:

```
select p1.nodeval
from path_table p1, path_index_table i1
where i1.path = 'a.b.c.d.'
and i1.pid = p1.pid
and p1.nodetype = 3;
```

This query checks for nodes corresponding to the path "a.b.c.d" that also have the appropriate node type to contain a node value (e.g., nodetype= "3"), and returns the value of those node(s) .

Consider the following XPath expression which contains a "[id=1]" predicate:

```
XPath: /a/b[id="1"]/c
```

This XPath expression can be translated to the following SQL statement that queries against the Path_Table and Path_Index_Table:

```
select p3.nodeval
from path_table p1, path_index_table i1,
path_table p2, path_index_table i2,
path_table p3, path_index_table i3
where i1.path = 'a.b.'
and i1.pid = p1.pid
and p2.docid = p1.docid
and p2.startpos > p1.startpos
and p2.startpos < p1.endpos
and p2.nodeval = '1'
and p2.pid = i2.pid
and i2.path = 'a.b.@id.'
and p3.docid = p1.docid
and p3.startpos > p1.startpos
and p3.startpos < p1.endpos
and p3.pid = i3.pid
and i3.path ='a.b.c.'
and p3.nodetype = 3;
```

Consider the following XPath expression, which includes the “//” symbol:

XPath: //c

The “//” symbol specifies selection of all the descendents of the document root.

A “like” operator can be used to evaluate this type XPath expression. This following is an example SQL statement that can be used to queries against the Path_Table and Path_Index_Table for this XPath expression:

```
select p1.nodeval
from path_table p1, path_index_table i1
where i1.path like '%c.'
and i1.pid = p1.pid
and p1.nodetype = 3;
```

The following XPath expression combines aspects of the previous three examples:

XPath: /a[//id>”1”]/b/c/d

This XPath expression can be translated to the following SQL statement:

```
select p3.nodeval
from path_table p1, path_index_table i1,
path_table p2, path_index_table i2,
path_table p3, path_index_table i3
where i1.path = 'a.'
and i1.pid = p1.pid
and p1.nodetype = 1
and p2.docid = p1.docid
and p2.nodetype = 3
and p2.nodeval > 1
and p2.pid = i2.pid
and i2.path like '%@id.'
and p3.docid = p1.docid
and p3.startpos > p1.startpos
and p3.startpos < p1.endpos
and p3.pid = i3.pid
and i3.path = 'a.b.c.d.'
and p3.nodetype = 3;
```

SYSTEM ARCHITECTURE OVERVIEW

The execution of the sequences of instructions required to practice the invention may be performed in embodiments of the invention by a computer system 1400 as shown in Fig. 7. In an embodiment of the invention, execution of the sequences of instructions required to practice the invention is performed by a single computer system 1400. According to other embodiments of the invention, two or more computer systems 1400 coupled by a communication link 1415 may perform the sequence of instructions required to practice the invention in coordination with one another. In order to avoid needlessly obscuring the invention, a description of only one computer system 1400 will be presented below; however, it should be understood that any number of computer systems 1400 may be employed to practice the invention.

A computer system 1400 according to an embodiment of the invention will now be described with reference to Fig. 7, which is a block diagram of the functional components of a computer system 1400 according to an embodiment of the invention. As used herein, the term computer system 1400 is broadly used to describe any computing device that can store and independently run one or more programs.

Each computer system 1400 may include a communication interface 1414 coupled to the bus 1406. The communication interface 1414 provides two-way communication between computer systems 1400. The communication interface 1414 of a respective computer system 1400 transmits and receives electrical, electromagnetic or optical signals, that include data streams representing various types of signal information, e.g., instructions, messages and data. A communication link 1415 links one computer system 1400 with another computer system 1400. For example, the communication link 1415 may be a LAN, in which case the communication interface 1414 may be a LAN card, or the communication link 1415 may be a PSTN, in which case the communication interface 1414 may be an integrated services digital network (ISDN) card or a modem.

A computer system 1400 may transmit and receive messages, data, and instructions, including program, i.e., application, code, through its respective communication link 1415 and communication interface 1414. Received program code may be executed by the respective processor(s) 1407 as it is received, and/or stored in the storage device 1410, or other associated non-volatile media, for later execution.

In an embodiment, the computer system 1400 operates in conjunction with a data storage system 1431, e.g., a data storage system 1431 that contains a database 1432 that is readily accessible by the computer system 1400. The computer system 1400 communicates with the data storage system 1431 through a data interface 1433. A data interface 1433, which is coupled to the bus 1406, transmits and receives electrical, electromagnetic or optical signals, that include data streams representing various types of signal information, e.g., instructions, messages and data. In embodiments of the invention, the functions of the data interface 1433 may be performed by the communication interface 1414.

Computer system 1400 includes a bus 1406 or other communication mechanism for communicating instructions, messages and data, collectively, information, and one or more processors 1407 coupled with the bus 1406 for processing information. Computer system 1400 also includes a main memory 1408, such as a random access memory (RAM) or other dynamic storage device, coupled to the bus 1406 for storing dynamic data and instructions to be executed by the processor(s) 1407. The main memory 1408 also may be used for storing temporary data, i.e., variables, or other intermediate information during execution of instructions by the processor(s) 1407.

The computer system 1400 may further include a read only memory (ROM) 1409 or other static storage device coupled to the bus 1406 for storing static data and instructions for the processor(s) 1407. A storage device 1410, such as a magnetic disk or optical disk, may also be provided and coupled to the bus 1406 for storing data and instructions for the processor(s) 1407.

A computer system 1400 may be coupled via the bus 1406 to a display device 1411, such as, but not limited to, a cathode ray tube (CRT), for displaying information to a user. An input device 1412, e.g., alphanumeric and other keys, is coupled to the bus 1406 for communicating information and command selections to the processor(s) 1407.

According to one embodiment of the invention, an individual computer system 1400 performs specific operations by their respective processor(s) 1407 executing one or more sequences of one or more instructions contained in the main memory 1408. Such instructions may be read into the main memory 1408 from another computer-usable medium, such as the ROM 1409 or the storage device 1410. Execution of the sequences of instructions contained in the main memory 1408 causes the processor(s) 1407 to perform the processes described herein. In alternative embodiments, hard-wired circuitry may be used in place of or in combination with software instructions to implement the invention. Thus, embodiments of the invention are not limited to any specific combination of hardware circuitry and/or software.

The term "computer-usable medium," as used herein, refers to any medium that provides information or is usable by the processor(s) 1407. Such a medium may take many forms, including, but not limited to, non-volatile, volatile and transmission media. Non-volatile media, i.e., media that can retain information in the absence of power, includes the ROM 1409, CD ROM, magnetic tape, and magnetic discs. Volatile media, i.e., media that can not retain information in the absence of power, includes the main memory 1408. Transmission media includes coaxial cables, copper wire and fiber optics, including the wires that comprise the bus 1406. Transmission media can also take the form of carrier waves; i.e., electromagnetic waves that can be modulated, as in frequency, amplitude or phase, to transmit information signals. Additionally, transmission media can take the form of acoustic or light waves, such as those generated during radio wave and infrared data communications.

In the foregoing specification, the invention has been described with reference to specific embodiments thereof. It will, however, be evident that various modifications and changes may be made thereto without departing from the broader spirit and scope of the invention. For example, the reader is to understand that the specific ordering and combination of process actions shown in the process flow diagrams described herein is merely illustrative, and the invention can be performed using different or additional process actions, or a different combination or ordering of process actions. The specification and drawings are, accordingly, to be regarded in an illustrative rather than restrictive sense.